# Efficient Upgrading in a Purely Functional Component Deployment Model

Eelco Dolstra

Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
`eelco@cs.uu.nl`

**Abstract.** Safe and efficient deployment of software components is an important aspect of CBSE. The Nix deployment system enables side-by-side deployment of different versions and variants of components, complete installation, safe upgrades, and safe uninstalls through garbage collection. It accomplishes this through a purely functional deployment model, meaning that the file system content of a component only depends on the inputs used to build it, and never changes afterwards. An apparent downside to this model is that upgrading "fundamental" components used as build inputs by many other components becomes expensive, since all of these must be rebuilt and redeployed. In this paper we show that binary patching between sets of components enables efficient deployment of upgrades in the purely functional model, transparently to users. Sequences of patches can be combined automatically to enable upgrading between arbitrary versions. The approach was empirically validated.

## 1 Introduction

An important aspect of Component-Based Software Engineering (CBSE) is the correct and efficient deployment of components after they have been developed [1]. This is often surprisingly hard. The main issues are dealing with side-by-side deployment of different versions or variants, isolation between components, ensuring complete component dependencies, and so on [2].

The Nix deployment system addresses these problems [3, 4]. The central idea is that each binary component is stored in isolation in the file system under a path name that contains a cryptographic hash of *all* inputs used to build the component, e.g., /nix/store/920e492a10af...-firefox-1.0. These inputs include (recursively) the component's build-time dependencies, build scripts, build parameters, platform, and so on.

The advantage is that we get variability support "for free": if two components are different in any way, they are stored in different locations in the file system. This isolation prevents undeclared build-time component dependencies. The hashes enable determination of run-time dependencies through a conservative pointer scanning approach [3]. This enables Nix to support side-by-side deployment of different versions and variants of components, complete installation, safe upgrades, and safe uninstalls through garbage collection.

However, there is a downside: *upgrading* becomes a much more resource-intensive operation. If we change any build-time input to a component, the hash of the component changes, and so it will need to be rebuilt and redeployed. This is the right thing to do, since the change to the input might actually matter in an observable way, i.e., we have obtained a new variant. However, for upgrades to "fundamental" components that are used directly or indirectly by many others, the cost of redeployment may be substantial. For instance, in the dependency graph of a typical Linux system, virtually all components depend on the GNU C Library (Glibc). An update to Glibc would therefore trigger a rebuild of all components in the system, similar to how a change to a common header file will cause massive recompilation in Make [5]. This is not a major issue since it takes place on the distributor's systems. Worse, however, is that in order to re-deploy the Glibc upgrade to end-users, each of them would need to download all rebuilt dependent components in addition to the new Glibc. This requires very substantial network resources; e.g., a small bug fix to Glibc might induce hundreds of megabytes worth of downloads.

In more conventional deployment models, upgrades are delivered as "destructive updates" that overwrite the older version of the component. This is more efficient but it short-circuits the dependency graph, inhibits rollbacks, and may break other installed components.

However, the inefficiency of deploying upgrades in Nix would suggest that the Nix system is not practical. *In this paper we show that it is.* By deploying *binary patches*, we can efficiently distribute new versions or variants of components to the end-user systems. For instance, a Glibc update typically causes a download of just a few hundred kilobytes in patches for around 150 components, a modest amount even on slow network connections.

*Contributions* The contributions of this paper are as follows.

- We show that the Nix deployment model can support efficient deployment of upgrades through the use of automatically generated binary patches that are transparently used by client machines.
- We introduce the technique of automatic *patch chaining* to relief the burden of having to generate patches between arbitrary releases.
- We show that patches between components can easily be produced, even in the presence of file or directory renames and moves, by producing deltas between *archives* of the components.

The techniques discussed in this paper have been implemented, and we discuss their effectiveness.

*Outline* The remainder of this paper is structured as follows. Section 2 gives a brief overview of the Nix system and motivates why we need binary patch deployment in Nix. Section 3 describes binary patch deployment in Nix, including the concept of patch chaining. The problem of selecting the right base components for patches is addressed in Section 4. We describe our experiences in Section 5. Related work is discussed in Section 6, and we conclude in Section 7.

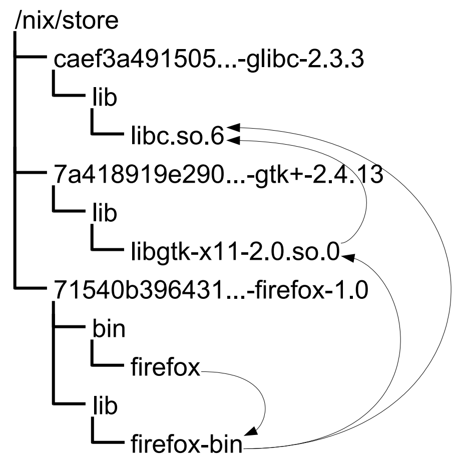
/nix/store
caef3a491505...-glibc-2.3.3
lib
libc.so.6
7a418919e290...-gtk+-2.4.13
lib
libgtk-x11-2.0.so.0
71540b396431...-firefox-1.0
bin
firefox
lib
firefox-bin

**Fig. 1.** The Nix store

## 2 Motivation

### 2.1 The Nix Deployment System

Nix is a system for software deployment [3, 4]. Its job is to build components, support their deployment to client machines, and manage the components on those clients. It has several important features:

- It supports component variability, allowing arbitrary side-by-side existence of multiple versions and variants (preventing the "DLL Hell"). Users or processes can have different "views" on the set of installed components.
- It helps ensure complete dependency specifications. Typical Unix package management systems such as RPM [6] require developers to specify their component's dependencies on other components. However, there is no assurance that such a specification is complete. This leads to incomplete deployment, i.e., references to missing components at run-time.
- It ensures consistency between components; e.g., that they are not removed from the system if they are required by other installed components.
- Components are built from a flexible component specification language — *Nix expressions* — supporting the concise specification of variability in components, such as domain features and dependencies.
- It supports binary deployment of components as an essentially transparent optimisation of source deployment, as explained below.

The central idea in the Nix system is that every component is stored in isolation in the *Nix store*. The store is a designated part of the file system (typically /nix/store), each subdirectory of which contains a component. An example of a number of Nix components on a Linux system is shown in Figure 1. The name of

```
{ stdenv, fetchurl, pkgconfig, gtk  # function arguments
, perl, zip, libIDL, libXi }:

assert libIDL.glib == gtk.glib;  # consistency requirement

stdenv.mkDerivation {  # the function result: a build action
  name = "firefox-1.0";

  builder = ./builder.sh;  # the build script
  src = fetchurl {  # the sources
    url = ftp://.../firefox/1.0/source/firefox-1.0-source.tar.bz2;
    md5 = "49c16a71f4de014ea471be81e46b1da8";
  };

  buildInputs = [pkgconfig gtk perl zip libIDL libXi];
}
```

**Fig. 2.** Nix expression for Firefox

each component directory contains, apart from a symbolic identifier of the component such as firefox-1.0, a unique hexadecimal number which is a *cryptographic hash* of *all* inputs involved in building the component.

For instance, for the Firefox component, the inputs include the operating system and platform for which we are building (e.g., i686-linux), the C++ compiler, the GNU C library, the GTK widget library, the X11 windowing system client libraries, the script that builds the component, the full sources of the component, and so on.

The arrows in Figure 1 denote file system references between components (note that it shows only a subset of Firefox's run-time dependencies; in reality Firefox has many more). For instance, the Firefox binary contains in its executable image the full path of the C and GTK libraries to be used at run-time (these are specified in the "RPATH" of Unix ELF executables [7]). That is, while those libraries are dynamically loaded at run-time, their locations are hard-coded into the components at build-time.

Nix components are built from *Nix expressions*, which is a simple functional language, a model well-suited for specifying components. Figure 2 shows the Nix expression for the Firefox component. This is actually a *function*[1] that accepts a number of arguments (e.g., gtk) and returns a *derivation*, which is Nix-speak for a component build action[2]. Likewise, there are Nix expressions specifying how to build the GNU C library (glibc), GTK, etc., when called with the appropriate arguments. Since these are all functions, to instantiate actual components, they must be called (i.e., *composed*). This is done in the Nix expression in Figure 3.

[1] I.e., it specifies *requires* interfaces of the component [2].

[2] A more complete description of the Nix expression language is given in [4] and in the Nix manual [8].

```
rec {

  firefox = (import ../applications/firefox) {  # function call
    inherit fetchurl stdenv pkgconfig gtk ...;  # arguments
  };

  gtk = (import ../development/libraries/gtk) {
    inherit fetchurl stdenv;
  };

  fetchurl = ...;
  stdenv = ...;
}
```

**Fig. 3.** Nix expression composing Firefox, GTK, etc.

When we evaluate the firefox attribute thus defined, Nix will recursively build all components insofar as they are not already present on the system.

As stated above, for each component Nix computes its path name in the Nix store by hashing all inputs used to build it. This is a recursive process: if the hash of any direct or indirect dependency of a component changes, the hash of the component itself will also change (since the hashes are 128-bit MD5 hashes, the chances of a hash collision are very slight indeed). For instance, if in the specification of the GTK component we change the source file from which it is built from release 2.2.4 to 2.4.13, the hash of Firefox will also change (Figure 4). Hence, both GTK and Firefox will be rebuilt.

Note that as we do so, any previous versions of GTK and Firefox are left untouched since they reside in a different location in the file system. This has important advantages. First, the previous installation of Firefox is unaffected. For instance, if the new GTK is not quite backwards compatible, it will not break the installed Firefox. Nix never relies on any notion of component interface compatibility, since those in practice cannot be trusted to completely specify the behaviour of the component. Only the implementation constitutes a full specification. Second, different users or processes can easily have different "activated" components (essentially, by having, for instance, different versions of Firefox in their PATH environment variables). Third, it enables efficient roll-back to previous versions if necessary, since the old version is still available until it is removed by running the Nix *garbage collector*.

Nix, as described above, implements a source deployment model (as do, e.g., FreeBSD [9] and Gentoo Linux [10]). That is, to deploy a component, we deploy to the clients the Nix expressions that describe how to build from source the component and its dependencies. While this is convenient for the developer, it is generally not appropriate for end-users since builds can consume substantial CPU, disk, and network resources. Also, it is inappropriate for closed source products. However, Nix can almost *transparently* support binary deployment

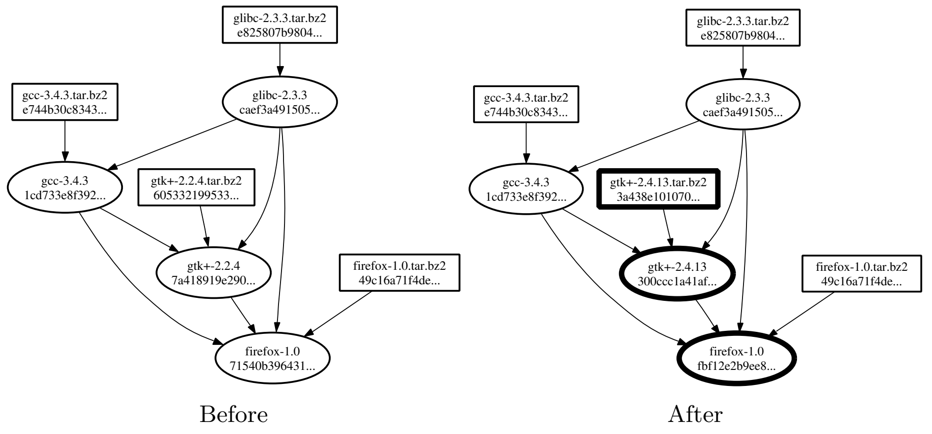

**Fig. 4.** Hash change propagation in the build-time dependency graph of the Firefox component after a GTK upgrade. Square nodes denote sources, with cryptographic hashes of the contents. Round nodes denote components, with store path hashes. Arrows indicate build-time dependencies.

thanks to the hashing scheme through its *substitute* mechanism, which works as follows. The component developer or distributor builds the Nix expression and uploads the resulting components to a repository accessible by the clients, typically a web server. A *manifest* on the server describes the available pre-built components, e.g,

```
{ StorePath: /nix/store/075931820cae...-firefox-1.0
  NarURL: http://server/075931820cae...-firefox-1.0.nar.bz2
  Size: 11480169 }
```

Then, when the client attempts to build the Nix expression, Nix will see that the path /nix/store/075931820cae...-firefox-1.0 that it wants to build is already present on the server. It will download the component from the URL given in the manifest and unpack it. On the other hand, if the path that it wants to build is not present on the server, if will fall back to building it from source, if possible.

## 2.2 Installing Upgrades

This paper is concerned with the efficient distribution of upgrades to components. Examples include the deployment of a security or other fix for Firefox or Glibc, or an ordinary version upgrade, such as Firefox 0.9 being updated to 1.0.

In conventional deployment models, upgrades are deployed "destructively". For instance, in RPM [6], we would just install a new Glibc RPM package that overwrites the old one. This however prevents side-by-side deployment of variants (what if some component *needs* the old Glibc because it is incompatible with the new one?), makes roll-backs much harder (essential in server environments), and is generally bad from a SCM perspective (since it becomes much harder to identify the current configuration). Also, such destructive upgrades only work with dynamic linking and other late-binding techniques; if the component has been statically linked into other components at build time, we must identify all affected components and upgrade them as well. This was for instance a major problem when a security bug was discovered in the ubiquitous Zlib compression library [11].

In Nix, on the other hand, as shown in Figure 4, the change to a component affects the hashes of all components that depend on it. That is, Nix has a *purely functional deployment model*: the "value" (i.e., file system contents) of a component only depends on the inputs used to build it, and never changes afterwards. This has two effects. First, all affected components must be rebuilt. This is exactly right, since the change to the dependencies may of course affect the derivates. This is the case even if interfaces haven't changed, e.g., in the case of statically linked libraries, smart cross-module inlining, changes to the compiler affecting the binary interface, and so on.

The second effect is that *all affected components must be re-deployed to the clients*, i.e., the clients must download and install each affected component. This is the scalability issue that this paper addresses. For instance, if we want to deploy a 100-byte bug fix to Glibc, almost all components in the system must be downloaded again, since at the very least the RPATHs of dependent binaries will have changed to point at the new Glibc. Depending on network characteristics, this can take many hours even on fast connections. Note that this is much worse than the first effect (having to rebuild the components), since that can be done centralised and only needs to be done once. The re-deployment, on the other hand, must be done for each client.

So why don't we just destructively update components in place, as is done in other deployment systems, e.g., by overwriting the old Glibc with the new one? The reason is that this violates the crucial deployment invariant that the hash of a path uniquely describes the component. (This is similar to allowing assignments in purely functional programming languages such as Haskell [12].) From a configuration management perspective, the hashes identify the configuration of the components, and destructive updates remove the ability to identify what we have on our system. Also, it destroys the component isolation property, i.e., that an upgrade to one component cannot cause the failure to another component. If an upgrade is not entirely backwards compatible, this no longer holds.

One might ask whether the relative difficulty (in terms of hardware resources, not developer or user effort) of deploying upgrades doesn't show that Nix is unsuitable for large-scale software deployment. However, Nix's advantages in supporting side-by-side variability, correct dependency, atomic rollbacks, and so on, in the face of a quasi-component model (i.e., the huge base of existing Unix packages) not designed to support those features, makes it compelling to seek a solution to the upgrade deployment problem within the Nix framework. The remainder of this paper shows such a solution.

## 3 Binary Patch Deployment

The previous section showed the explosion in the number of components to be redeployed in case of an update to a fundamental component such as Glibc. In this section we solve this problem by transparently deploying *binary patches* between component releases. For instance, if a bug fix to Glibc induces a switch from /nix/store/219a...-glibc-2.3.3 to /nix/store/ff9c...-glibc-2.3.3p1, then we compute the delta (the *binary patch*) between the contents of those paths and make the patch available to clients. We also do this for all components depending on it. Subsequently the clients can apply those patches to the old version to produce the new version. As we shall see in Section 5, patches for components affected by a change to a dependency are generally very small.

A binary patch describes a set of edit operations that transforms a *base* component stored at path $X$ in the Nix store into a *target* component stored at path $Y$. Thus, if a client needs path $Y$ *and* it has path $X$ already installed, then it can speed up the installation of $Y$ by downloading the patch from $X$ to $Y$, copy $X$ to $Y$ in the Nix store, and finally apply the patch to $Y$.

This fits nicely into Nix's substitute mechanism used to implement transparent binary deployment. We just extend its download capabilities: rather than doing a full download, if a patch is available, we download that instead. Manifests can specify the availability of patches. For instance,

```
patch {
  StorePath: /nix/store/5bfd71c253db...-firefox-1.0
  NarURL: http://server/52c036147222...-firefox-1.0-to-1.0.nar-bsdiff
  Size: 357
  BasePath: /nix/store/075931820cae...-firefox-1.0
}
```

describes a 357-byte patch from the Firefox component shown in the previous section (stored at BasePath) to a new one (stored at StorePath) induced by a Glibc upgrade. If a patch is not available, or if the base component is not installed, we fall back to a full download of the new component; or even a local build if no download is available.

### 3.1 Binary Patch Creation

There are many off-the-shelf algorithms and implementations to compute binary deltas between two arbitrary files. We used the bsdiff utility [13] because it produced the smallest patches compared to others (see Section 6).

However, the components in the Nix store are arbitrary directory trees. How do we produce deltas between directories trees? A "simple" solution would be to compute deltas between corresponding regular files (i.e., with the same relative path in the components) and distribute all deltas together. The full contents of all new files in the target should also be added, as well as a list describing file deletions, changes to symlink contents, etc. Files not listed would be assumed to be unchanged.

This method is both complex and has the severe problem of not following renames. For instance, the Firefox component stores most of its files in a subdirectory lib/firefox-*version*. The method described above would fail to patch, e.g., lib/firefox-0.9/libmozjs.so into lib/firefox-1.0/libmozjs.so since the path names do not correspond; rather, the latter file would be stored in full in the patch. Hence, patching would not be very effective in the presence of renames.

There is however a much simpler and more effective solution: we take the patch between *archive files* of the components. For instance, we can produce an uncompressed Zip or Tar file[3] containing the full contents of the directory trees of the two components, and compute the binary delta between those files. This automatically takes renames, deletions, file type changes, etc. into account, since these are just simple changes within the archive files. To apply a patch, a client must also create an archive of the base component, apply the binary patch to it, and unpack the resulting archive into the target path.

For instance, for the example above, the base archive will at some point contain a filename lib/firefox-0.9/libmozjs.so followed by the contents of that file, while the target archive will contain a filename lib/firefox-1.0/libmozjs.so followed by its contents, which may or may not be the same as the original file. The binary delta algorithm will just emit an edit operation that changes the first file name into the second, followed by the appropriate edit operations for the file contents. It does not matter whether the position of the file in the archive has changed: contrary to delta algorithms like the standard diff tool, bsdiff can handle re-ordering of the data.

## 3.2 Patch Chaining

It is generally infeasible to produce patches between every pair of releases of a set of components. The number of patches would be $O(n^2m)$, where $n$ is the number of releases and $m$ is the number of components. As an example, consider the *Nix Packages collection* (Nixpkgs). It is a set of existing Unix components ranging from GCC to Firefox. Pre-releases of Nixpkgs are made on every commit to its version management repository, which typically is several times a day. Developers and users can stay up-to-date by subscribing to a *channel*, which is just a convenience mechanism for updating Nix expressions on client machines.

Since pre-releases appear so often, we cannot feasibly produce patches between each pair of pre-releases. So as a general policy we only produce patches between immediately succeeding pre-releases. For instance, given releases 0.7pre1899, 0.7pre1928 and 0.7pre1931, we produce patches between 0.7pre1899 and 0.7pre1928, and between 0.7pre1928 and 0.7pre1931. This creates a problem, however: suppose that a user has Firefox from 0.7pre1899 installed, and Firefox changed in

[3] In actuality, we produce a "Nar file", which is Nix's archive file format. Nar files, unlike Tar or Zip files, have a *canonical form*: there is a single, uniquely defined archive for the contents of a directory. For instance, directory entries are always stored in the same order. This minimises the chances of a patch failing to apply due to version differences in the archiving tool.

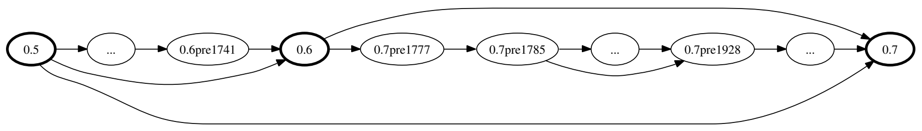

**Fig. 5.** Patch set created between Nixpkgs releases. Arrows indicate the existence of a patch set.

both succeeding releases, then there would be no patch that brings the user up-to-date.

The solution is to automatically *chain* patches, i.e., using a *series* of available patches $X_1 \rightarrow ... \rightarrow X_n \rightarrow Y$ to produce path $Y$. In the example above, we have a Firefox component installed that can be used as the base path to a patch in the 0.7pre1928 release, to produce a component that can in turn serve as a base path to a patch in the 0.7pre1931 release.

However, such patch sequences can eventually become so large that they approach, or become larger than, full downloads. In that case we can "short-circuit" the sequence by adding patches between additional releases. Figure 5 shows an example sequence of patches between releases thus formed. Here, patch sets are produced by directly succeeding pre-releases, and between any successive stable releases. An additional "short-circuit" patch set between 0.7pre1785 and 0.7pre1928 was also made.

In the presence of patch sets between arbitrary releases, it is not directly obvious which sequence of patches or full downloads is optimal. To be fully general, the Nix substitute downloader runs a shortest path algorithm on a directed acyclic graph that, intuitively, represents components already installed, available patches between components, and available full downloads of components. Formally, the graph is defined as follows:

- The nodes are the store paths for which pre-built binaries are available on the server, either as full downloads or as patches, plus any store paths that serve as bases to patches. There is also a special start node.
- There are three types of edges:
  - *Patch edges* between store paths that represent available patches. The edge weight is the size of the patch (in bytes).
  - *Full download edges* from start to a store path for which we have a full download available. The edge weight is the size of full download.
  - *Free edges* from start to a store path representing that a store path is already available on the system. The edge weight is 0.

We then find the shortest path between start and the path of the requested component using Dijkstra's shortest path algorithm. This method can find any of the following:

- A sequence of patches transforming an already installed component into the requested component.

- A full download of the requested component.
- A full download of some component $X$ which is then transformed using a sequence of patches into the requested component. Generally, this will be longer than immediately doing a full download of the requested component, but this allows one to make *only* patches available for upgrades.

Above, edge weight was defined as the size in bytes of downloads. We could take other factors into account, such as protocol/network overhead per download, the CPU resources necessary to apply patches, and so on. For instance, on a reasonably fast connection, a full download might be preferable over a long sequence of patches even if the combined byte count of those patches is less than the full download.

## 4 Base Selection

To deploy an upgrade, we have to produce patches between "corresponding" components. This is intuitively simple: for instance, to deploy a Glibc upgrade, we have to produce patches between the old Glibc and the new one, but also between the components depending on it, e.g., between the old Firefox and the new one. However, a complication is that the dependency graphs might not be isomorphic. For instance, components may have been removed or added, dependencies moved, component names changed (e.g., Phoenix to Firebird to Firefox), and so on. Also, even disregarding component renames, simply matching by name is insufficient because there may be multiple component instances with the same name (e.g., builds for different platforms).

The *base selection problem* is the problem, when deploying a set of target components $\mathbb{Y}$, of selecting from a set of base components $\mathbb{X}$ a set of patches $(X, Y) \in (\mathbb{X} \times \mathbb{Y})$ such that the probability of the $X$s being present on the clients is maximised *within* certain resource constraints.

Clearly, we could produce patches between all $X$s and $Y$s. This policy is "optimal" in the sense that the client would always be able to select the absolutely shortest sequence of patches. However, it is infeasible in terms of time and space since producing a patch takes a non-negligible amount of time, and most such patches will be large since they will be between unrelated components (for instance, patching Acrobat Reader into Firefox is obviously inefficient).

Therefore, we need to select some subset of $(\mathbb{X} \times \mathbb{Y})$. The solution currently implemented is heuristical: we use a number of properties of the components to guess whether they "match" (i.e., are conceptually the "same" component). Indeed, the selection problem seems inherently heuristical for two reasons. First, there can be arbitrary changes between releases. Second, we cannot feasibly produce all patches to select the "best" according to some objective criterion.

Possible heuristics include the following:

- *Same component name.* This is clearly one of the simplest and most effective criteria. However, there is a complication: there can be multiple components with the same name. For instance, Nixpkgs contains the GNU C Compiler gcc at several levels in the dependency graph (due to bootstrapping). Also, it contains two components called firefox — one is the "real thing", the other is a shell script wrapper around the first to enable some plugins. Finally, Nixpkgs contains the same components for multiple platforms.

– The *weighted number of uses* can be used to disambiguate between components at different bootstrapping levels such as GCC mentioned above, or disambiguate between certain variants of a component. It is defined for a component at path $p$ as follows:

$$w(p) = \sum_{q \in \text{users}(p)} \frac{1}{r^{d(q,p)}}$$

where users($p$) is the set of components from which $p$ is reachable in the build-time dependency graph, i.e., the components that are directly or indirectly dependent on $p$; where $d(q,p)$ is the unweighted distance from component $q$ to $p$ in the build-time dependency graph; and where $r \geq 1$ is an empirically determined value that causes less weight to be given to "distant" dependencies than to "nearby" dependencies.

For instance, in the Nixpkgs dependency graph, there is a "bootstrap" GCC and a "final" GCC, the former being used to compile the latter, and the latter being used to compile almost all other packages. If we were to take the unweighted number of uses ($r = 1$), then the bootstrap GCC would have a slightly higher number of uses than the final GCC (since any component using the latter is indirectly dependent on the former), but the difference is too small for disambiguation — such a difference could also be caused by the addition or removal of dependent components. However, if we take, e.g., $r = 2$, then the weighted number of uses for the final GCC will be almost twice as large. This is because the bootstrap GCC is at least one step further away in the dependency graph from the majority of components, thus halving their contribution to its $w(p)$.

Thus, if the ratio between $w(p)$ and $w(q)$ is greater than some empirically determined value $k$, then components $p$ and $q$ are considered unrelated, and no patch between them is produced. A good value for $k$ would be around 2, e.g., $k = 1.9$.

– *Size* of the component. If the ratio between the sizes of two components differs more than some value $l$, then the components are considered unrelated. A typical value would be $l = 3$; even if components differing in size by a factor of 3 *are* related, then patching is unlikely to be effectual. This trivial heuristic can disambiguate between the two Firefox components mentioned above, since the wrapper script component is much smaller than the real Firefox component.

– *Platform.* In general, it is pointless to create a patch between components for different platforms (e.g., Linux and Mac OS X), since it is unlikely that a client has components for a different platform installed.

## 5 Experience

We have implemented the binary patch deployment scheme described above in the Nix system, and used it to produce patches between 50 subsequent (pre-) releases of the Nix Packages collection[4]. Base components were selected on the basis of matching names, using the size and weighted number of uses heuristics described in the previous section to disambiguate between a number of components with equal names. The use of patches is automatic and completely transparent to users; an upgrade action in Nix uses (a sequence of) patches if available and applicable, and falls back to full downloads otherwise. In this section we provide some data to show that the patching scheme succeeds in its main goal, i.e., reducing network bandwidth consumption in the face of updates to fundamental components such as Glibc or GCC to an "acceptable" level.

We computed for each pair of subsequent releases how large an upgrade using *full downloads* of changed components would be, versus downloading *patches* to changed components. Also, the average and median sizes of each patch for the changed components (or full download, if no patch was possible) were computed. New top-level components (e.g., applications introduced in the new release) were disregarded. Table 1 summarises the results for a number of selected releases, representing various types of upgrades. File sizes are in bytes unless specified otherwise. Omitted releases were typically upgrades of single leaf components such as applications. An example is the Firefox upgrade in revision 0.6pre1702.

Efficient upgrades or patches to fundamental components are the main goal of this paper. For instance, release 0.7pre1980 upgraded the GNU C Compiler used to build all other components, while releases 0.7pre1820 and 0.7pre1977 provided bug fixes to the GNU C Library, also used at build-time and run-time by all other components. The patches resulting from the Glibc changes in particular are tiny: the median patch size is around 440 bytes. This is because such patches generally only need to modify the RPATH in executable and shared libraries. The average is higher (around 4K) because a handful of applications and libraries statically link against Glibc components. Still, the *total* size of the patches for all components is only 598K and 743K, respectively — a fairly trivial size even on slow modem connections.

On the other hand, release 0.6pre1489 is not small at all — the patch savings are only 60.5%. However, this release contained many significant changes. In particular, there was a major upgrade to GCC, with important changes to the generated code in all components. In general, compilers should not be switched lightly. (If *individual* components need an upgraded version, e.g., to fix a code generation bug, that is no problem: Nix expressions, being a functional language, can easily express that different components must be built with different compilers.) Minor compiler upgrades need not be a problem; release 0.7pre1980, which featured a minor upgrade to GCC, has a 98.1% patch effectiveness.

[4] The releases are available at http://catamaran.labs.cs.uu.nl/dist/nix, and the Nix expressions from which they were generated at https://svn.cs.uu.nl:12443/viewcvs/trace/nixpkgs/trunk/.

| Release | Comps. changed | Full size | Total patch size | Savings | Avg. patch size | Median patch size | Remarks |
|---|---|---|---|---|---|---|---|
| 0.6pre1069 | 27 | 31.6M | 162K | 99.5% | 6172 | 898 | X11 client libraries update |
| 0.6pre1489 | 147 | 180M | 71M | **60.5%** | 495K | 81K | Glibc 2.3.2 → 2.3.3, GCC 3.3.3 → 3.4.2, many other changes[5] |
| 0.6pre1538 | 147 | 176.7M | 364K | 99.8% | 2536 | 509 | Standard build environment changes |
| 0.6pre1542 | 1 | 9.3M | 67K | 99.3% | 67K | 67K | Firefox extensions/profiles bug fix |
| 0.6pre1672 | 26 | 38.0M | 562K | 98.6% | 22155 | 6475 | GTK updates |
| 0.6pre1702 | 3 | 11.0M | 190K | 98.3% | 63K | 234K | Firefox 1.0rc1 → 1.0rc2 |
| 0.7pre1820 | 154 | 188.6M | 598K | 99.7% | 3981 | 446 | Glibc loadlocale bug fix |
| 0.7pre1931 | 1 | 1164K | 45K | 96.1% | 45K | 45K | Subversion 1.1.1 → 1.1.2 |
| 0.7pre1977 | 153 | 196.3M | 743K | 99.6% | 4977 | 440 | Glibc UTF-8 locales patch |
| 0.7pre1980 | 154 | 197.2M | 3748K | 98.1% | 24924 | 974 | GCC 3.4.2 → 3.4.3 |

**Table 1.** Statistics for patch sets between selected Nixpkgs releases and their immediate predecessors

Patch generation is a relatively slow process. For example, the generation of the patch set for release 0.7pre1820 took 49 minutes on a 3.2 GHz Pentium 4 machine with 1 GB of RAM running Linux 2.4.26. The bsdiff program also needs a large amount of memory; its documentation recommends a working set of at least 8 times the base file. For a large component such as Glibc, which takes 46M of disk space, this works out to 368M of RAM.

A final point not addressed previously is the disk space consumption of upgrades. A change to a component such as Glibc will still cause every component to be duplicated on disk, even if they do no longer have to be downloaded in full. However, after an upgrade, a user can run the Nix garbage collector that safely and automatically removes unused components. Nonetheless, as an optimisation, we observe that many files in those components will be exactly the same (e.g., header files, scripts, documentation, JAR files). Therefore, we implemented a tool that "optimises" the Nix store by finding all identical regular files in the store, and replacing them with hard links [14] to a single copy. On typical Nix stores (i.e., subject to normal evolution over a period of time) this saved between 15–30% of disk space. While this is useful, it is not an order of complexity change as is the case with the amount of bandwidth saved using patches.

## 6 Related Work

Binary patching has a long history, going back to manual patching of binaries on mainframes in the 1960s, where it was often a more efficient method of fix-

[5] First release since 0.6pre1398.

ing bugs than recompiling from source. Binary patching has been available in commercial patch tools such as .RTPatch, and interactive installer tools such as InstallShield. Most Unix binary package managers only support upgrades through full downloads (SuSE's "Patch RPMs" include full copies of all changed files). Microsoft recently introduced binary patching in Windows XP Service Pack 2 as a method to speed up bug fix deployment [15].

A method for automatically computing and distributing binary patches between FreeBSD releases is described in [16]. It addresses the additional complication that FreeBSD systems are often built from source, and the resulting binaries can differ even if the sources are the same, for instance, due to timestamps being stored in files. In the Nix patching scheme we guard against this possibility by providing the MD5 hash of the archive to which the patch applies. If it does not apply, we fall back to a full download. In general, however, this situation does not occur because patches are obtained from the same source as the original binaries.

In Nix, the use of patches is completely hidden from users, who only observe it as a speed increase. In general, deployment methods often require users to figure out what files to download to install an upgrade (e.g., hotfixes in Windows). Also, if sequences of patches are required, these must be applied manually by the user, unless the distributor has consolidated them into a single patch. The creation of patches is often a manual and error-prone process, e.g., figuring out what components to redeploy as a result of a security bug like [11]. In our approach, this determination is automatic.

The bsdiff program [13] that Nix uses to generate patches is based on the *qsufsort* algorithm [17]. In our experience bsdiff outperformed methods such as ZDelta [18] and VDelta [19, 20], but a comparison of delta algorithms is beyond the scope of this paper. An overview of some delta algorithms is given in [19].

The problem of keeping derivates consistent with sources and dependency graph specifications occurs in all build systems, e.g., Make [5]. To ensure correctness, such systems must rebuild all dependent objects if some source changes. If a source is fundamental, then a large number of build actions may be necessary. So this problem is not unique in any way to Nix. However, the problems of build systems affect developers, not end users, while Nix is a *deployment* system first and foremost. This is why it is important to ensure that end users are not affected by the use of a strict update propagation semantics.

## 7 Conclusion

In a previous ICSE paper [3] we introduced the purely functional deployment model underlying Nix, where components are stored in isolation in paths in the file system that contain a hash of all build-time inputs used to construct the component, and argued that this has substantial advantages for the safe deployment of software. However, as we noted then, the downside to such a model is that updates to fundamental components require all components depending on them to be updated also. In this paper we have shown that using binary patch deployment, Nix's functional deployment model can in fact efficiently and transparently support such operations.

*Acknowledgements* This research was supported by CIBIT | SERC and the NWO Jacquard program. The author would like to thank Martin Bravenboer and Eelco Visser for commenting on drafts of this paper.

## References

1. Carzaniga, A., Fuggetta, A., Hall, R.S., Heimbigner, D., van der Hoek, A., Wolf, A.L.: A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado (1998)
2. Szyperski, C.: Component technology—what, where, and how? In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003). (2003) 684–693
3. Dolstra, E., Visser, E., de Jonge, M.: Imposing a memory management discipline on software deployment. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), IEEE Computer Society (2004) 583–592
4. Dolstra, E., de Jonge, M., Visser, E.: Nix: A safe and policy-free system for software deployment. In Damon, L., ed.: 18th Large Installation System Administration Conference (LISA '04), Atlanta, Georgia, USA, USENIX (2004) 79–92
5. Feldman, S.I.: Make—a program for maintaining computer programs. Software—Practice and Experience **9** (1979) 255–65
6. Foster-Johnson, E.: Red Hat RPM Guide. John Wiley and Sons (2003)
7. TIS Committee: Tool Interface Specification (TIS) Executable and Linking Format (ELF) Specification, Version 1.2. http://www.x86.org/ftp/manuals/tools/elf.pdf (1995)
8. TraCE Project: Nix deployment system. http://www.cs.uu.nl/groups/ST/Trace/Nix (2005)
9. FreeBSD Project: FreeBSD Ports Collection. http://www.freebsd.org/ports/ (2005)
10. Gentoo Project: Gentoo Linux. http://www.gentoo.org/ (2005)
11. Adler, M., Gailly, J.: Zlib advisory 2002-03-11. http://www.gzip.org/zlib/advisory-2002-03-11.txt (2002)
12. Peyton Jones, S., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2004)
13. Percival, C.: Binary diff/patch utility. http://www.daemonology.net/bsdiff/ (2003)
14. Stevens, W.R.: Advanced Programming in the UNIX Environment. Addison-Wesley (1993)
15. Microsoft Corporation: Binary delta compression. Whitepaper (2002)
16. Percival, C.: An automated binary security update system for FreeBSD. In: Proceedings of BSDCON '03, USENIX (2003)
17. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Technical Report LU-CS-TR-99-214, Lund University (1999)
18. Trendafilov, D., Memon, N., Suel, T.: zdelta: An efficient delta compression tool. Technical Report TR-CIS-2002-02, Polytechnic University (2002)
19. Hunt, J.J., Vo, K.P., Tichy, W.F.: Delta algorithms: An empirical analysis. ACM Transactions on Software Engineering and Methodology **7** (1998) 192–214
20. Korn, D., Vo, K.: vdelta: Differencing and compression. In Krishnamurthy, B., ed.: Practical Reusable UNIX Software. John Wiley & Sons (1995)